Distributed Development of Automotive Real-time Systems based on Function-triggered Timing Constraints

Oliver Scheickl, Christoph Ainhauser, Michael Rudorfer

BMW Car IT, Munich, Germany

Abstract: This paper proposes a new model-driven approach to develop automotive real-time systems. Instead of constraining implementation-driven timing properties – like offsets, periods or the like – for software, our approach uses so-called function-triggered timing constraints as basis of system configuration. These constraints are implementation-independent. The main focus is how such kinds of constraints can be used to derive abstract configuration boundaries or budgets for the different development teams in a so-called distributed development environment. Distributed development is a typical strategy in the automotive domain, where different teams are involved in the development of modern car functions. Our approach thus decouples the implementation and timing configuration work of the teams. The main contributions of our work are a methodology for distributed development of automotive real-time systems, a special timing model as basis for the methodology, and an algorithmic approach to break down function-triggered timing constraints to local requirements for the involved teams.

Keywords: Automotive Real-time Systems, Distributed Development, Contract-based Design, Timing Model

1. Introduction

Today's automobile features include an increasing number of functions that are realized by electronics and software. These functions are typically provided by interactive distributed real time systems. The development of these vehicle electrical systems is a complex task mainly for the following reasons:

1. Functions are often distributed across the system and may involve several electronic control units (ECUs), sensors, actuators and communication busses for their execution.
2. Each ECU may be involved in the realization of many different functions. This leads to a mutual influence of the functions on each ECU.
3. Subsystems are often developed by different teams and suppliers and have to be integrated by the car manufacturer (OEM).
4. The ECUs realize an increasing number of functions. This leads to a higher degree of integration on each ECU.
5. The distributed functions often have to fulfil stringent timing constraints to function properly.

A significant increase in outstanding innovations is expected for future automobile generations. The industry and research community is searching for methods to cope with the increasing complexity of automotive system design.

To be prepared for the increasing overall complexity of automotive embedded systems major automotive OEMs and tier-1 suppliers founded the AUTOSAR development partnership in the year 2003. Today, many OEMs, suppliers as well as software and hardware companies participate in the partnership [10]. The main goal of the initiative is to define a common development methodology and standardized software architecture for ECUs with well defined module interfaces [1]. As a basis for this, a comprehensive structural system model comprising software components, their communication, basic software, software mapping to ECUs etc. can be described in a standardized formal way. This information can then be exchanged across car manufacturer and suppliers if necessary, i.e. across different development teams.

With the latest version 4.0 of AUTOSAR also the required timing behaviour of the system is addressed [1]. An early proposal for a timing-augmented AUTOSAR specification was presented in [3]. Now, a generic approach has been developed within the AUTOSAR partnership to attach a so called timing description to an AUTOSAR system model [13]. In this way, the structural system model can be extended to directly carry timing constraint information. The timing model in our work utilizes the generic AUTOSAR timing specification concept. Our methodology presented in this work fits to the AUTOSAR approach.

2. Motivation

2.1 Function-triggered Timing Constraints

Different kinds of timing constraints exist in practice. In our work we concentrate on the following three typical ones:

1. A latency constraint constrains the latency between two successive events (e.g. "from reading the acceleration sensor until the execution of the damper actuator").

2. A triggering constraint constrains the period of a cyclic event ("the damper control function should be triggered with a period of 10 milliseconds").
3. A synchronization constraint relates the temporal synchronicity of events ("all four dampers of the car have to react synchronously").

As these examples illustrate, the concept of “events” is essential for our timing model. Section 3.2 details the timing model and the usage of events.

In our understanding, timing constraints are either a result of physics or of requirements from a customer's perspective. The constraints in general refer to a function, regardless of its implementation in the car. Therefore we call such kind of timing constraints *function-triggered timing constraints*. An implementation by means of hardware and software must then be chosen in a way that all function-triggered timing constraints are always fulfilled. An implementation contains a lot of so called *timing properties* that influence the timing behaviour. Focusing on software, such properties are execution times, schedules or bus frame configurations. The resulting timing behaviour must be correct with respect to the function-triggered timing constraints. Thus the properties itself must be chosen with respect to these constraints. Timing properties are implementation details and shall not be the target of invariant, global timing constraints. In other words, the timing constraints in our methodology (see Section 3.3) are not implementation-driven but, as mentioned before, function-triggered.

2.2 Roles in Distributed Automotive Development

In the automotive industry frequently a distributed development strategy is observed. That is, the overall vehicle electrical system is developed by many different teams. The teams have different roles in the process and therefore bear responsibility for different parts of the system. Three main roles are identified that are important for our proposed methodology.

The *system designer* designs and finally integrates the vehicle electrical system. The system designer (typically an OEM or first-tier supplier) specifies the system’s network (sensors, controllers, actuators, and busses), the software architecture with its software components (SWCs), and their mapping onto the network’s ECUs. The SWCs are part of the implementation of all desired car functions. The ECUs are then actually developed by *ECU integrators* (typically first-tier suppliers). This role integrates the SWCs and basic software of an ECU according to the system designer’s specification. As one of AUTOSAR’s proposed benefits, software components may be delivered by third party *software component suppliers*, which is the third role.

It is the system designer’s task to integrate all ECUs to a functioning system. It is the ECU integrator’s task to integrate all software components on an ECU according to its specification. To avoid inconsistencies and misunderstandings in the collaboration between the three mentioned roles, a common specification format is required. Therefore AUTOSAR offers its standardized software architecture and its standardized exchange format. Since release 4.0 also timing constraints can be added to the specification.

With these three roles – which actually may be represented by many people – we can identify two typical collaboration use cases in the automotive domain. The first one is the integration of an ECU into a system. System designer and ECU integrator have to collaborate in this use case. The second one is rather new for the automotive industry and mainly enabled by the AUTOSAR approach. Due to the standardized software component interfaces now also the integration of third-party software components into an ECU is possible. Therefore, system and ECU integrators have to collaborate with the software component supplier.

2.3 Timing Responsibilities of the Roles

In Section 2.1 we mentioned that various timing properties influence the timing behaviour of the system and thus the fulfilment of its function-triggered timing constraints. Each timing property is influenced by one role. In the following we describe this responsibility assignment.

As the system designer is responsible for the system, he must also be aware of the system's function-triggered timing constraints. In our approach these constraints are considered as the basis of system development (from a real-time viewpoint). The system designer chooses the network of communicating software components, the hardware topology and - as one of the most important design decisions with influence on the timing behaviour - the mapping of software components to ECUs. Bus design also is the system designer's responsibility. Bus design covers the mapping of data elements to bus frames and the frame scheduling.

The ECU integrator configures the basic software and application software components on an ECU according to the specification. Software components have to be mapped to operating system tasks with an appropriate task schedule. The schedule as implementation property of course highly influences the timing behaviour.

The software component supplier cannot directly influence the runtime timing behaviour of software components as it is integrated by the ECU integrator. However, he can provide component-based timing properties like execution times, required execution periods or a required execution order.

2.4 Problem Statement

Functions can be distributed across the system. Function-triggered timing constraints thus can lead to mutual timing dependencies of different teams and roles in a distributed development environment. There are many examples for timing dependencies in distributed development: In a synchronous FlexRay network the ECU and bus schedules can be tightly coupled if they use the same time base [5]. The change of the bus schedule can influence data availability for tasks scheduled on connected ECUs. In an asynchronous CAN network all sending ECUs may influence the sending behaviour of each other dynamically. Also, a certain software component's implementation leads to a concrete execution time when integrated on an ECU. The synchronization of events on different ECUs implies the synchronization of the schedules of different ECU integrators. Our work concentrates on this kind of collaboration dependencies regarding a system's timing. To tackle this problem, we define a special timing model and a methodology for distributed development of automotive real-time systems based on function-triggered timing constraints. Furthermore we develop algorithms to generate timing requirements for the different development teams according to the function-triggered timing constraints.

In our work we focus on the role *system designer*. The system designer has the responsibility to ensure the system's correct timing behaviour. That means he must coordinate the previously mentioned timing dependencies of the various teams and roles. ECU integrators and especially software component developers not necessarily know the function-triggered timing constraints. Therefore they need a clear specification of the desired component's timing requirements. The formal model of AUTOSAR (system model plus generic timing model) offers a valuable basis for our approach. The AUTOSAR standard does not answer the question how the fulfilment of function-triggered timing constraints can be guaranteed in distributed system development. Our work addresses exactly this issue.

2.5 Related Work

Our work shares some similar ideas of a general approach called contract-based design [12]. In computer science this is an approach to develop software systems, especially large systems, with distributed development. The interfaces of components shall precisely be described in a formal and verifiable way. For different aspects of functional or non-functional behaviour components give assertions (in our context guarantees) based on some assumptions (in our context requirements), i.e. a contract among components. This idea has been adopted by different domains. [2] also proposes a contract-based design approach motivated by an automotive example. Safety and timing properties (latency, triggering) can be expressed in a language similar to Linear Temporal Logic. The approach however neither considers function-triggered timing constraints nor an iterative finding of appropriate requirements. Rich Components [4] are another concept to add non-functional requirements to component models. End-to-end timing in terms of a “path through a system” is also considered in a system-wide timing analysis approach by [6]. The model described in [7] assumes the knowledge of all system details like task mapping, bus and task scheduling. The decomposition of an end-to-end constraint to component requirements is not part of the model. The authors in [8] discuss the applicability of AUTOSAR in the context of system-level integration, component interfaces and portability.

3. Approach

3.1 Modelling a System

Our work follows the model-based development approach. A “system" in our understanding is a static system model that contains all information needed for our approach. The system model is similar to other models known from literature and industry (e.g. AUTOSAR [1]) but only contains the elements and attributes that are important for our task. Its scope is the static structure (components, ports, etc.) of the system, not the system's dynamic behaviour, i.e. its expected runtime actions and interaction.

3.1.1 Software Structure Model

The software of the system is modelled as a set of interconnected *software components*. A SWC is either a composition that can contain other SWC or an atomic SWC. An atomic SWC contains a set of runnables that model pieces of executable code. Two special types of atomic SWCs exist: sensor and actuator SWCs can be used to model sensor and actuator access. Furthermore a SWC can have *required and provided ports* on which they can receive and send *data elements* from and to other SWCs via *connectors*. The complete software structure is assumed as one composition that contains a network of interconnected atomic SWC. For simplification we neglect component hierarchies.

3.1.2 Hardware Topology Model

The hardware topology of the system is modelled as a network of ECUs connected to a communication bus. In our context, an ECU solely serves as container for SWCs. There are two types of busses supported, namely CAN and FlexRay. For FlexRay networks an ECU can either be synchronized to the bus or unsynchronized.

3.1.3 System Model

The model of a complete system consists of a software structure, a hardware topology and, most notably, the software mapping. A software mapping describes which atomic SWC is executed on which ECU. As a result, all connectors of the software model can now be identified as local or remote connectors. For local connectors immediate data transmission is assumed. The data elements of remote connectors however have to be transmitted over a communication bus.

3.1.4 Communication Model

Each data element that is remotely exchanged between SWCs is represented by a signal. A set of such signals is mapped into a frame that is transmitted over the communication bus. Every frame has some scheduling attributes. For a CAN frame a message ID (priority) can be modelled. For a FlexRay frame the slot ID and communication cycles can be modelled (actually an offset to the overall cycle start). The so called communication matrix models which ECU reads or writes which frame or signal respectively. This information can be gained from the software structure and its mapping, i.e. from a system model.

3.2 Timex Model

Our Timex model is used to attach a so called *timing extension* to a system model. A timing extension is used to describe the static timing relations within a given system. First of all we explain the basic principle of events and event chains on which our Timex model (**Tim**ing **Ex**tension) is based (see Figure 1). After that we explain Timex itself.

Our Timex model is based on the concept of so called *observable events* and *event chains* [3]. Observable events represent a condition of certain system behaviour. In our context the following events exist: *external* events at a sensor or actuator, *data element received* event at a required port, *data element* sent event at a provided port, *signal queued for transmission* if a signal is ready to be transmitted and *signal sent to bus* if a signal has been transmitted. An event chain refers to two observable events, namely the chain's stimulus and the chain's response. The semantics of such a chain is that the response occurs as causal consequence of the stimulus. Stimulus and response thus also have a clear temporal order (or dependency order [11]).

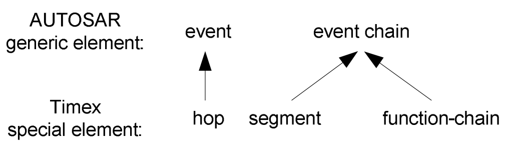


Figure 1: Relation of AUTOSAR and Timex elements

The structure of Timex consists of the following elements.

3.2.1 Hand Over Points

In our methodology, some of the many existing observable events in a system are of special interest. These are the events on the border of two responsibilities of different teams, or roles. They mark the places in the model where data is handed over from one team to another. Therefore in Timex these events can be characterized as *hand over points* (hop).

3.2.2 Function Timing and Function-chains

Function timing is used to capture all function-triggered timing constraints as explained in Section 2.1. Function timing consists of a set of so-called *function-chains*. This is an event chain as described in Section 3.2 with a special semantics: it has a stimulus hop and a response hop and models an end-to-end timing dependency of a function. It is not necessary to already know exactly what happens in between these two hops.

3.2.3 Timing Constraints

Furthermore, function timing contains a set of *timing constraints* that constrain the timing behaviour of the function-chains. As these constraints are independent of a concrete implementation (i.e. of a system) function timing must be attached to some kind of function model. AUTOSAR, which is our basic system model, does not provide the concept of a “function”. Instead, we use a composition without atomic SWCs inside to model a function.

A latency timing constraint refers to one function-chain and constrains the minimum and maximum latency between each of that chain's stimulus and response occurrences. A synchronization constraint refers to two or more function-chains and constrains the synchronicity of the chains' stimulus or response hops. A triggering constraint refers to one function-chain and constrains the periodic execution of that function-chain.

3.2.4 System Timing and Segments

After function timing, a certain *system timing* can be defined. If a composition is used in a system as described in Section 3.1.3 a lot of implementation details are known (SWC mapping, bus type). According to our methodology the system designer also can determine which parts of the overall system are developed by which team. Thus he can identify system timing hops additionally to the ones already known from the function timing. All these hops can now be used to model a set of so-called *segments*. A segment is a special event chain with a stimulus and a response hop. In a consistent Timex model every function-chain is refined by an arbitrary number of segments without gaps and also without overlaps. One segment can be used for the refinement of many function-chains. In other words, a segment belongs to one team, but one segment can be part of several function-chains. System timing consists of all additional hops and all segments and it is attached to a system model as its timing extension.

A complete Timex model of a system contains the function-triggered timing constraints of all function-chains modelled as function timing and the implementation-specific segments modelled as system timing. We call the complete representation of a function-chain by means of segments the function-chain's *segmentation*. An example Timex model (function-chain and its segmentation) with all relevant model elements is depicted in Figure 2.

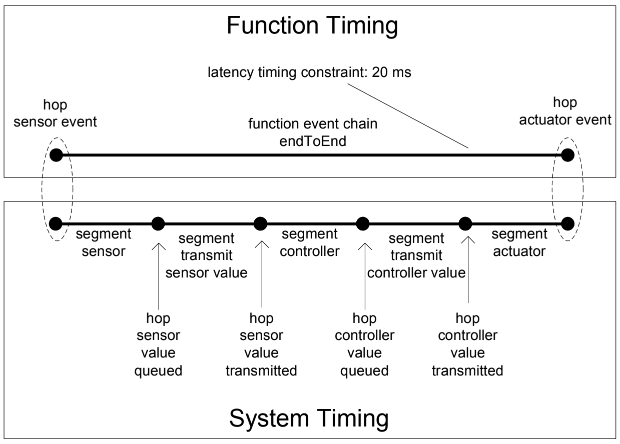

Function Timing
latency timing constraint: 20 ms
hop sensor event
function event chain endToEnd
hop actuator event
segment sensor
segment transmit sensor value
segment controller
segment transmit controller value
segment actuator
hop sensor value queued
hop sensor value transmitted
hop controller value queued
hop controller value transmitted
System Timing

Figure 2: Example Timex model

3.3 Methodology

Our methodology, described in this section, focuses on the system designer role described in Section 2.2. It is mainly following an idealized top-down approach for system development. In practice of course system development is more complex and is subject to many restrictions and prerequisites. Here, the methodology is sufficient to explain the basic problem, our proposed approach, and our solution.

Figure 3 depicts our methodology. It consists of six steps. The first three steps are performed manually by engineering work. Steps *4*, *5* and *6* are performed automatically, i.e. using algorithms. Steps *5* and *6* are performed iteratively. In the following each step is described in detail.

3.3.1 Define Function Timing

The first step is the definition of the function timing model (Section 3.2.2) where all function-triggered timing constraints are collected (Section 3.2.3). Function-triggered timing constraints are implementation-independent, so no system model is necessary in this step. A function model however is needed to attach function timing. As described in Section 3.2.3 we use an empty composition for this purpose.

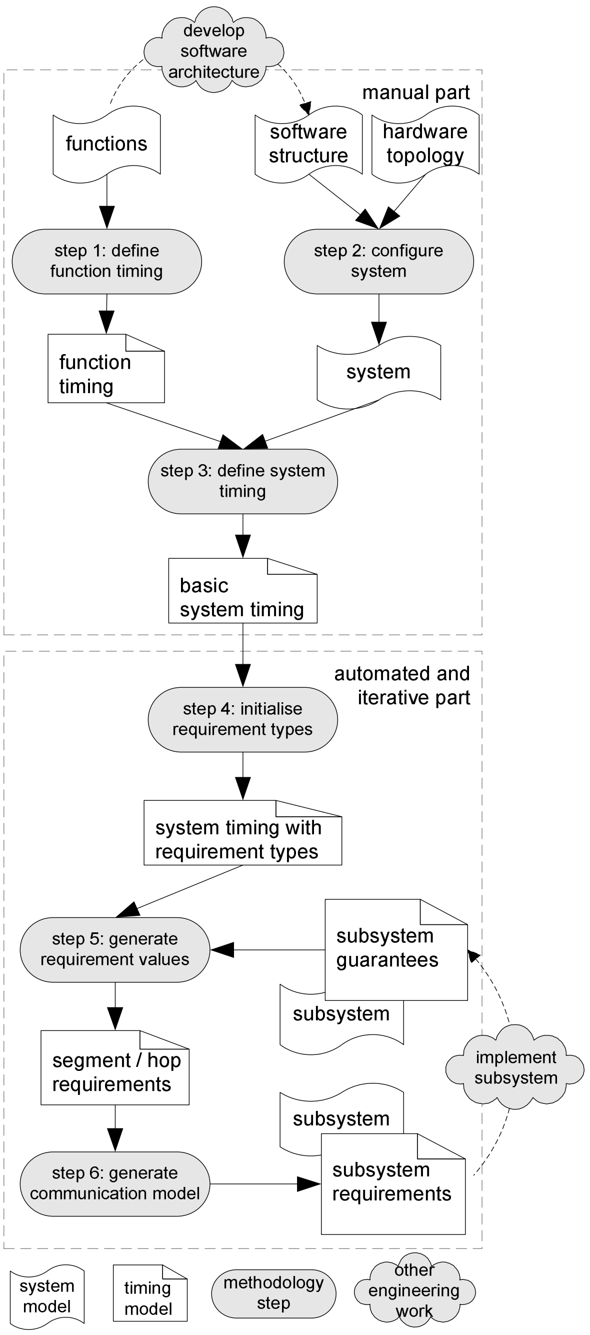

develop software architecture
manual part
functions
software structure
hardware topology
step 1: define function timing
step 2: configure system
function timing
system
step 3: define system timing
basic system timing
automated and iterative part
step 4: initialise requirement types
system timing with requirement types
step 5: generate requirement values
subsystem guarantees
subsystem
implement subsystem
segment / hop requirements
subsystem
step 6: generate communication model
subsystem requirements
system model
timing model
methodology step
other engineering work

Figure 3: Methodology for distributed development based on function-triggered timing constraints

Each observable event offered by the function model or the software structure model can be characterized as a hop. They can be used for the definition of function-chains, which in turn have function timing constraints. In function timing these hops typically are "end-events" of signal paths like external events and runnable entity terminated events. A function timing model is always valid in the context of the function model to which it is attached. However, the function model can be used in several system models and the according function timing can thus be re-used. This concept makes function timing independent from an implementation (i.e. concrete system).

3.3.2 Configure System

System configuration means to map the atomic SWCs of a software structure model onto ECUs of a hardware topology model. The design process of developing a software architecture (i.e. software structure model) for the desired functional scope is neglected here. Furthermore, a signal is created for each data element sent over a remote connector. The output of this step is the model of a system. We assume that the system designer knows enough details of the complete system.

3.3.3 Define System Timing

A system model and the according function timing model are input for the definition of the system timing. In this step additional communication-related and SWC-related events can be characterized as hops because now also the software structure, software mapping and the resulting communication is known. All additional hops of the system timing must be on the signal path between the function timing end-to-end hops. We assume that the system designer as central role knows the responsibilities within the complete system and thus can identify which events separate two teams. Each segment created that way belongs to one responsibility, i.e. to one development team. The output is the Timex system timing “graph”, consisting of hops (vertices) and segments (edges).

3.3.4 Initialise Requirement Types

The system timing model carries the requirement and guarantee values later in the methodology. Therefore the required types of requirements for segments and hops must be initialised once per system. The goal is to be able to express all function timing constraints by system timing requirements. This step can be automated. We explain its details in Section 4.1.

3.3.5 Generate Requirement Values

As mentioned earlier function-triggered timing constraints are independent from concrete implementation details. A special challenge in the proposed methodology is the tracing of design decisions and their influence on the generation of requirements starting from function-triggered timing constraints. One design decision is the software mapping. It influences which communication will be remote and thus influences which requirements are necessary for communication. Furthermore, the type of hardware topology as design decision influences the type of requirements to be generated. We made the observation that such kind of design decisions influence the way how a function-triggered timing constraint is mapped to requirements.

The goal of this methodology step is to generate a set of timing requirements for certain segments and hops of the system timing model (see step *4*). The requirements must be chosen in a way that the fulfilment of all function-triggered constraints is ensured if all requirements are fulfilled by their guarantees. We will focus on that in Section 5.

3.3.6 Generate Communication Model

As last step the communication model as described in Section 3.1.4 is generated. We assume that the system designer is responsible for the configuration of the system's communication. The communication model then is the basis for the work of all other teams. Primarily, three tasks have to be performed in this step. First, all signals have to be grouped in frames. Second, a communication matrix has to be generated using the communication information gathered from the software structure and the software mapping. Third, a scheduling of the frames has to be generated according to the communication type (FlexRay or CAN) and with respect to the requirements generated before. The resulting communication model must fulfil all communication-related requirements.

3.3.7 Iterative Steps

After the last methodology step each involved team implements its subsystem, as depicted in the cloud in Figure 3. A subsystem is either an ECU, if the segments belong to a team in the role of an ECU integrator, or a single SWC, if the segments belong to a team in the role of a software component supplier. For each subsystem there are segments with timing requirements to be met. The teams must implement and configure their subsystem according to these requirements, if possible (see Section 5). Otherwise the system's correct timing behaviour cannot be ensured.

When the subsystems have been implemented by the teams, timing guarantees for each segment are given back to the system designer. The guarantees are evaluated with respect to their requirements. If all requirements are met by the guarantees no action is required and the system is expected to fulfil all its function-triggered timing constraints. Otherwise the guarantees are input for the next iteration of steps *5* and *6* of the methodology and a new set of requirement values must be generated.

Today, this negotiation process takes place in an informal way. The Timex model and our proposed methodology formalize it.

4. Problem Analysis

In this section we analyze the problem of steps *4* and *5* of our methodology depicted in Figure 3. Thus, we assume that a system model is given (step *2*) and function timing (step *1*) as well as system timing (step *3*) has been specified using Timex.

We reemphasise our terminology. Timing constraints are system-wide, function-triggered and invariant. They must be fulfilled by an implementation. A timing requirement always refers to a single segment or hop. Requirements are generated and valid for one methodology iteration. They are accompanied by an according guarantee that must fulfil the requirement.

In step *5* of our methodology a set of independent timing requirements for segments and hops shall be generated. These requirements must be fulfilled by the team in charge, which in turn delivers an appropriate timing guarantee. The problem of step 4 is to transform the timing constraints into timing requirements for segments and hops, such that the fulfilment of all requirements by their according guarantees results in the fulfilment of all timing constraints ("getting independently verifiable requirements"). This is done once per system.

If the guarantees do not fulfil all requirements, a new iteration is initiated. So the problem of step 5 is that in iteration n+1 the guarantees for the requirements of iteration n must be considered. That means the new requirements shall not be generated arbitrary or randomly, but with respect to the before generated ones ("getting valid requirement values").

A solution to the problem of step *4* ensures that correct timing is evaluated on the level of single requirements instead of system-wide constraints. A solution to the problem of step *5* ensures that valid requirement values are found.

The rest of this section is devoted to step *4*. Section 5 outlines considerations to step *5*.

4.1 Types of Segments

A system timing model consists of segments that form function-chains. These are specified in the according function timing. With the set of possible events as hops (Section 3.2) and our considered roles and use cases (Section 2.2) there are certain types of possible segments. Table 1 lists all possible segment types for the use case ECU integration.

Segment type	**Stimulus Hop**	**Response Hop**
sensor-to-bus	external	signal queued
transmission	signal queued	signal transmitted
over-ecu	signal transmitted	signal queued
bus-to-actuator	signal transmitted	external

Table 1: Segment types for the use case "ECU Integration"

For the second use case SWC integration all segment types except transmission can possibly be refined. Now the path over a SWC within the respective segment becomes visible in system timing. Table 2 lists the additional segment types.

Segment type	**Stimulus Hop**	**Response Hop**
receive-data	signal transmitted	data on required port
over-swc	data on required port	data on provided port
send-data	data on provided port	signal transmitted

Table 2: Additional segment types needed for the use case "SWC Integration"

Certain design decisions influence the segmentation of function-chains, i.e. what types of segments are needed for their segmentation. The first decision, of course, is the set of atomic SWCs. The second is the software mapping. The third are the use cases that influence what segment types are needed (see Tables 1 and 2). These design decisions are already considered in methodology step *3*.
Once all necessary segment types for the system timing have been determined and the system timing is complete a valid segmentation of all function-chains is given.

4.2 Mapping of Constraints to Requirements

As described in the beginning of section 4, function-constraints shall be expressed as a set of individual requirements for segments and hops. Each of the three possible constraint types latency, synchronization and triggering (see Section 3.2.1) must be mapped to requirements. We define the following three possible types of requirements:

- A segment can have a latency requirement. The segment must then have a certain minimum and maximum latency.
- A hop can have an offset requirement. The hop must have a minimum and maximum offset ("distance") to a certain reference hop.
- A hop can have a triggering requirement. The hop's event must then have a certain period.

Note that there is no such thing as a synchronization requirement possible on the granularity of segments and hops. Synchronization always requires several objects to be referenced. This is not possible for segments and hops, because their requirements shall be independent from other segments. That is, synchronization constraints (for function-chains) as well as the other two constraint types must be mapped to the requirement types given above.

In the next Section we show this mapping using a simple example.

4.3 Example

Consider the example given in Figure 2. The function chain “endToEnd” is already segmented to five segments, based on the software mapping design decision and the appropriate use cases. As you can see in the example, the applied use case is ECU integration of a sensor ECU, a controller ECU, and an actuator ECU. Thus, in this example we consider the ECUs as “black boxes” with no third-party software components on them. The system designer is responsible for two transmission segments. The ECU integrators of the three ECUs each are responsible for their ECU segment.

The communication type as design decision influences the type of requirements that segments and hops of the system timing have to fulfil. The requirement types are different for a CAN network compared to a FlexRay network.

In the following we analyse the two different sets of requirement types for the latency constraint in Figure 2. The requirement type sets are given in Table 3.

Segment	**CAN**	**FlexRay**
sensor	latency	latency
transmit sensor value	latency	triggering + offset for stimulus and response hop
controller	latency	latency
transmit controller value	latency	triggering + offset for stimulus and response hop
actuator	latency	latency

Table 3: Requirement types needed for the same function-chain in a CAN and a FlexRay system

4.3.1 CAN

For a CAN network we assume that the control flow is given by the data flow. In our case that means the triggering of a stimulus hop’s event triggers the response hop’s event. The latency constraint can therefore be mapped to five latency requirements, one for each segment. The sum of all min/max latency requirement values must be greater/smaller than the latency constraint’s min/max value. However, every team only has to ensure the fulfilment of its requirement, regardless of the overall constraint (similar to a divide and conquer approach). For example, the system designer must ensure that the sensor value and controller value transmission is performed within its required latency.

4.3.2 FlexRay

In case of a time-triggered bus like FlexRay the control flow is not directly given with the data flow since data transmission is triggered by the progress of time. It must be specified when data is expected to be queued for transmission and is transmitted. Therefore an offset requirement is used. Besides the minimum and maximum values, an offset requirement also specifies a so called “source hop”. The actual target hop must have the specified offset relative to the source hop. In a FlexRay network, the source hop typically is the cycle start of the network. The transmission latency is expressed my means of relative offsets. Because of the cyclic repetition of FlexRay communication, also a triggering must be specified for each queued and transmitted hop. Therefore an additional triggering requirement is necessary to map the initial latency constraint to a FlexRay network. Here we only want to show, which requirement types are necessary, not which values they must have. This is investigated in Section 5.

4.4 Fulfilment of Requirements with Guarantees

Methodology step *5* includes the consideration of timing guarantees in the requirement generation process to avoid random requirement generation. Therefore we first define, in which case a timing guarantee fulfils a timing requirement.

- A latency requirement specifies a minimum and a maximum latency, i.e. an interval. The guaranteed interval must be within the required interval.
- An offset requirement specifies a minimum and a maximum offset, i.e. an interval. Again, the offset guarantee interval must be within the offset requirement interval.
- An event triggering requirement for a hop specifies a period. The guaranteed period must be the same as the required period.

5. Collaboration Scenarios

In this section we want to outline our approach of iterative requirement value generation. As depicted in our methodology in Figure 3, new requirement values must be generated if not all requirements are fulfilled by their guarantees.

To illustrate our iterative approach we will stick to our example of Figure 2 and assume a CAN network, i.e. a requirement setup as shown in the second column of Table 3.

5.1 One Function-chain Example

In the first simple scenario, we assume only one function-chain (Table 4) with a maximum latency constraint. The minimum value is neglected here because the concept is the same.

Three cases are possible for a requirement value:

1. If a latency requirement is exactly fulfilled by its guarantee (requirement value equals guarantee value) everything is fine.

2. If a guarantee value is even smaller than its requirement value then the requirement is fulfilled and additional buffer is gained.

3. If a guarantee value is greater than its requirement value then there is a problem. A new iteration must be initiated to generate a new set of requirement values.

Segment	**Iteration n**		**Iteration n+1**	
	Requ.	Guar.	Requ.	Guar.
sensor	8	4	4	4
transmit sensor val.	2	2	2	2
controller	6	4	4	4
transmit controller	2	2	2	2
actuator	2	6	6	6
Sum	**20**	**18**	**18**	**18**

Table 4: Collaboration scenario, one function-chain

In Table 4 the transmission segments exactly fulfil their requirements in iteration *n*. The actuator segment violates its requirement. Though, the sensor and controller segments have smaller guarantee values. This buffer can be used to grant a more relaxed requirement for the actuator in iteration *n+1*. We call this process *horizontal shifting* (shifting buffer along a chain).

5.2 Two Function-chains Example

In this more complex scenario of Table 5 we assume we have two (or more) such function-chains. For simplicity we now assume they are independent, i.e. they don't have common segments.

Segment	**Iteration n**		**Iteration n+1**	
	Requ.	Guar.	Requ.	Guar.
sensor 1	8	6	6	6
transmit sensor 1	2	2	2	2
controller 1	4	4	6	6
transmit controller 1	2	2	2	2
actuator 1	4	4	4	4
Sum	**20**	**18**	**20**	**20**
sensor 2	8	8	8	8
transmit sensor 2	2	2	2	2
controller 2	4	6	4	4
transmit controller 2	2	2	2	2
actuator 2	4	4	4	4
Sum	**20**	**22**	**20**	**20**

Table 5: Collaboration scenario, two function-chains

In chain 1 the sensor segment's guarantee is smaller than its requirement so there is a buffer of 2 in iteration *n*. Chain 2 however has a violated requirement at its controller segment. In iteration *n+1* the buffer of chain 1 can be used for chain 2 indirectly, because both controller segments are on the same ECU. We assume, if one segment's requirement is relaxed, another segment's requirement on the same resource can be tightened. We call this process vertical shifting (shifting buffer along a resource).

5.3 Generalization of the Problem

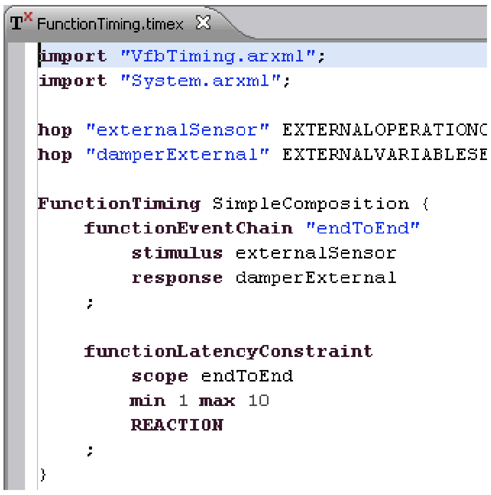
The two examples showed our basic ideas how a functioning timing configuration can be found using an iterative process. Therefore the use of buffers is essential. For a real-world system with several such chains and multiple development teams a complex framework of constraints arises. That means that in fact it is not trivial to determine valid requirement values. In particular, a "manual approach" like in these examples will be inapplicable. We use constraint logic programming (CLP) techniques to formalize all dependencies of requirement values and to generate new values during an iteration of methodology steps *5* and *6.* Our constraint system sets up different classes of rules that constrain the resulting requirement values:

- Basic, system-specific rules, e.g. the offset of a transmission hop must be smaller than its period
- Rules to ensure that all function-triggered constraints are fulfilled with the requirements (independence of requirements), e.g. by summing the segment latencies of chains
- Rules that express horizontal and vertical shifting as explained in Sections 5.1 and 5.2
- Optimization rules, e.g. maximize the latency of a function chain to save resources

A constraint solver can then be used to determine requirement values that fulfil all of these CLP constraints.

6. Prototype Implementation

Our Methodology is implemented as a prototype tool. It is based on Artop [9]. Artop is an Eclipse-based community approach to deliver a common platform for AUTOSAR tools. It offers basic functionalities, like an AUTOSAR meta-model implementation, model and workspace management or basic model editors. Artop is the fundament for the implementation of our prototype.



```
import "VfbTiming.arxml";
import "System.arxml";

hop "externalSensor" EXTERNALOPERATIONC
hop "damperExternal" EXTERNALVARIABLESE

FunctionTiming SimpleComposition {
    functionEventChain "endToEnd"
        stimulus externalSensor
        response damperExternal
    ;

    functionLatencyConstraint
        scope endToEnd
        min 1 max 10
        REACTION
    ;
}
```

Figure 4: Function timing example

We use Artop's plug-in concept to implement all Timex functionalities as plug-ins. The plug-ins offer:

- The Timex meta-model (Section 3.2)
- A textual Timex editor to model Function Timing and System Timing (Figure 4)
- A graphical Timex viewer to display the structure of function-chains, segments and hops graphically (Figure 5)
- An implementation of methodology step *4*
- CLP algorithms as example implementation of methodology step *5*
- A simple communication generator (step *6*)
- A converter to export generated Timex requirements to a standard AUTOSAR 4.0 timing model and vice versa

Our prototype covers the complete methodology as described in Section 3.3

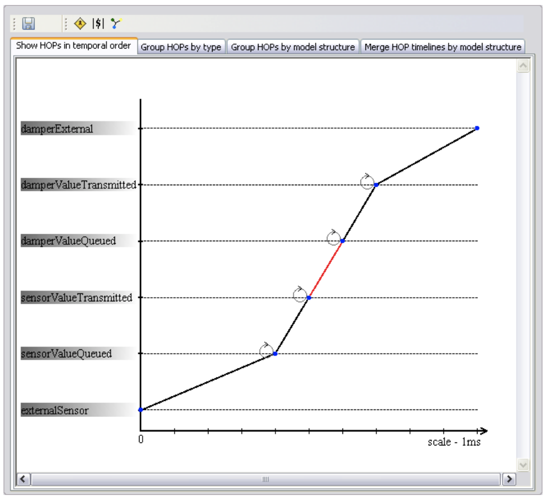


Figure 5: Graphical Timex viewer

7. Summary

In this article we presented Timex, a development methodology and timing model for automotive systems. The focus of Timex is a distributed development of distributed automotive real-time systems. Our methodology focuses on the role system designer in such a distributed development environment. This role integrates the overall system and must ensure the fulfilment of its timing constraints. Timing constraints are considered as function-triggered. We propose a timing model for the methodology and an approach to map function-triggered timing constraints to requirements to be fulfilled by the involved teams. In an iterative process, requirement values are determined, such that a) the implementation of all teams can fulfil their requirements with guarantees and b) the fulfilment of all requirements implies the fulfilment of all function-triggered timing constraints. The methodology is illustrated with some examples. Currently we are developing a formal description of our approach.

8. References

[1] AUTOSAR Development Partnership: "*Automotive Open System Architecture*", http://www.autosar.org.

[2] Brunel et. al: "*SoftContract: an Assertion-Based Software Development Process that Enables Design-by-Contract*", Proc. of DATE'04, 2004.

[3] O. Scheickl, M. Rudorfer: "*Automotive real time development using a timing-augmented AUTOSAR specification*", Proceedings of ERTS, 2008.

[4] W. Damm et. al.: "Boosting Re-use of Embedded Automotive Applications Through Rich Components" In Proc. of Foundations of Interface Technologies, 2005.

[5] S. Reichelt, O. Scheickl, G. Tabanoglu: "*The Influence of Real-time Constraints on the Design of FlexRay-based Systems*", DATE'09, 2009.

[6] N. Feiertag et. al.: "*A Compositional Framework for End-to-End Path Delay Calculation of Automotive Systems under Different Path Semantics*", *Proceedings of the IEEE Real-Time System Symposium (RTSS)*, 2008.

[7] R. Henia et. al.: "*System Level Performance Analysis - the SymTA/S Approach*", IEE Proc. of Computers and Digital Techniques, 2005.

[8] Heinecke et. al.: *"Software components for reliable automotive systems"*, Proc. of Date'08, 2008.

[9] M. Rudorfer et. al.: *"Artop – an ecosystem approach for collaborative AUTOSAR tool development"*, Proc. of ERTS^2, 2010.

[10] S. Fürst et. al.: *"AUTOSAR – A Worldwide Standard is on the Road"*, Proc. of 14th International Congress Electronic Systems for Vehicles, 2009.

[11] M. Broy: *"Time and Causality in Interactive Distributed Systems"*, Summer School, 2008.

[12] B. Meyer: *"Applying Design by Contract"*, IEEE Computer Society Press Volume 25, 1992.

[13] AUTOSAR Development Partnership: *"Specification of Timing Extensions"*, http://autosar.org/download/R4.0/AUTOSAR_TPS_TimingExtensions.pdf, 2010.